SI 486H, Unit 2: Architecture

Daniel S. Roche (roche@usna.edu)

Spring 2016

This unit looks at the various ways in which computers can generate random numbers, which of course is fundamental for everything we'll be doing in this course! We'll look both at hardware-based methods as well as software-based generators.

1 Entropy and Information Theory

Claude Shannon was working at Bell Labs in 1948 on how to efficiently encode digital information for efficient and effective transmission. In developing a mathematical theory of encoding and decoding, he wanted a way to *quantify* information in some meaningful way. So he decided to invent the field of information theory. This theory has many applications in engineering and computer science, and as we'll see there is also an important connection to randomness.

For us, information will always be measured in bits. You know a bit as a discrete kind of thing which can be either a one or a zero. And it certainly is that. But we're also going to abstract the idea a little bit away from the physical realm and allow things like half of a bit. Essentially, a *bit* is a unit of measurement just like an inch or a gram.

1.1 Prof email

Now consider some message, say, an email from Prof. X to student Y saying

```
Sounds good, see you Wednesday at 3:30.
```

This could, for example, be a response to a request for an EI schedule. The question is, how much *information* is in this message? As I've written it, there are 39 characters in the body of the message, and if each one is encoded as a single ASCII byte, that means 312 bits. If you consider the entire message with headers and everything, it might be more like 1000 bits or more. Are there really 1000 bits of information in this message?

The answer is, *maybe* (but probably not). Think about what information is actually being communicated here. First, assuming this email is a response to another email, the headers and subject probably don't tell anyone anything useful that they didn't already know. So we're back to the 312 bits of the message itself.

Then within that message, most of the text is not really crucial. It seems reasonable that the same *information* could be transmitted more simply by writing yes Wed. 3:30. Now that's just 13 characters, or 104 bits of data. In fact, since I've only used standard English letters and punctuation, we could use just 6 bits per character, for 78 bits.

Can we do better? Sure! Maybe the original message (that this is a response to) said something like "Can we meet after 6th period this week?" In that case, the time doesn't give any new information, and the real information content is the fact that there is a meeting, and the day of the week. If that's all that's being communicated, how many possibilities were there? Well, we have a "yes" answer with one of 7 days in the week, or a "no" answer, for a total of 8 possibilities. So we could make a scheme like saying 0 means "no" and 1-7 mean that day of the week, and encode this entire message as a number from 0-7. As you know, that takes just $\lg 8 = 3$ bits to store. Much smaller!

We could keep playing these games with this example, but the point I'm trying to make is that *you can't determine the information content just by looking at the message.* What determines the information content is the *possibilities* of what that message could have been. The greater the variation in what the message *could* have said, the greater the information content.

1.2 Weather and likeliness

Now consider another email situation: you get a message every morning telling you the weather forecast for the day. This has the predicted temperature and cloudiness/precipitation.

Most days, the weather looks pretty much like yesterday, maybe with some small variations. In the month of December 2015 in Annapolis, it was pretty much either high around 55 and sunny, or high around 55 and raining. So really, most of the time, the weather forecast just tells us whether it will be sunny or rainy. Since there are just two choices, the *information content* in these "typical" messages is something like 1 bit.

But those aren't the only two possibilities for the forecast! Maybe one morning, the forecast says "100 degree heat and tornadoes". That would be quite a surprise - and it is *exactly this same reason* that such a message would have much more information than the sunny or rainy message that we got the other days.

In general, the principle here is that **the lower the probability of something happening, the more information in conveys**.

1.3 Quantifying information

Enough general talk - let's get some numbers! We'll start with what we understand well: if we have 2^k possible outcomes, and each of them happens with equal probability, then each outcome conveys the same amount of information. And we know that we need exactly k bits to specify one of 2^k possibilities. So we'll say that if $\Pr[x] = \frac{1}{2^k}$, then $\mathrm{Inf}[x] = k$ bits.

Now let's generalize this: what if the event x has probability p, where p is just some number between 0 and 1? We just substitute p for $\frac{1}{2^k}$ in the equation above to get the definition of information content:

> **Definition**: Information content
>
> If an event x occurs with probability p, then the information conveyed in case x actually happens is

$$\lg \frac{1}{p} = -\lg p$$

This definition comes from Shannon's original theory and correlates with our intuitions about information: the less likely something is to happen, the more information it conveys.

1.4 Entropy

We are now ready for the conclusion of our discussion of information theory, the definition of *Shannon entropy.* The term "entropy" in chemistry or physics generally refers to the amount of disorder or chaos (or *randomness*!) in a system. In the context of information theory, it means something rather precise: the expected amount of information conveyed by a single event.

Using the definition above, and the definition of expected value from before, this is not too hard to write down.

Definition: Shannon entropy

If n events are possible, with probabilities $p_1, \ldots, p_n$, then the entropy in any single event's occurrence, in bits, is

$$\sum_{i=1}^{n} p_i \lg \frac{1}{p_i} = -\sum_{i=1}^{n} p_i \lg p_i$$

For example, if we are talking about flipping a "fair" coin with probability one half for heads or tails, the entropy in a single flip is

$$\sum_{i=1}^{2} \frac{1}{2} \lg 2$$

which works out to just 1 bit. This makes sense; in fact, the entropy in *any* situation that has k equally-likely events is exactly $\lg k$ bits. More interesting things happen when the events are not equally likely, as you will see in the problems.

2 Randomness

At this point I hope that the connection between randomness and information theory is becoming more clear. In fact, the meaning of the term *entropy* in chemistry and physics is sort of about chaos or disorder - randomness in physical systems! That this is tightly related to the concept of *information* is a little more surprising, but it has to do with the concept of information as a *distinction* between different alternatives. If something is not sufficiently chaotic or unpredictable, then it can't be used to make decisions. And that's exactly what our random number algorithms are all about!

2.1 Measuring randomness

Dilbert and XKCD make some important points about randomness. Seriously, think about what those cartoons are saying.

If you go to http://www.random.org you can see that they sample atmospheric radio noise - which is supposed to be random - and generate random numbers from it. You can actually buy random numbers if you want! But how can we tell whether the numbers from their website are any more or less random than the ones from the Dilbert cartoon?

Unfortunately, the short answer is that *we can't say anything definitively.* In fact, it's impossible to look at a sequence of numbers and say whether or not it's random, since it's always *possible* that that sequence was generated randomly. This is the same reason that we can't look at a text file and say how much information it contains - it has to do with the *set of possibilities* that that text file or those numbers came from.

But in practice, we measure the randomness (or entropy) of various sources all the time, through a variety of statistical tests. Since this isn't a statistics class, we're not going to go into all the details; I recommend anyone to read Knuth's book (mentioned above) if you want to learn about some of the tests with names like Frequency Test, Serial Test, Gap Test, Poker Test, Coupon Collector's Test, Collision Test, Spectral Test,... the list goes on and on.

The basic principle behind all these statistical tests is the following: *if* the sequence were generated randomly, what is the probability that it would look the way it does? For example, if a stream of supposedly random bits started out with 100 1's in a row, we would say that this sequence is *probably* not random. That's because the probability that a random sequence starts with 100 1's is $1/2^{100}$, an astronomically small number. So while such a sequence *could* be randomly-generated, chances are heavily in favor of the opposite hypothesis.

2.2 True random number generators

From last week's reading, we discussed a few kinds of "true" random number generators that get used in practice. These all have the general flavor of observing some controlled electronic process, then recording the results and spitting out the bits. This is true both of devices that you can buy to connect to your computer as well as built-in ones like on the Ivy Bridge processor.

Besides such *physical* sources of randomness, people have also looked into human sources. Typical examples are the timing between your keystrokes, or of motion changes when you move your mouse around. It is exactly these sources that are used to populate the bits of the /dev/random stream on most modern Linux machines, for example.

Often, these "true" random sources are biased in some way. For example, they might spit out a "1" bit 75% of the time, and a "0" bit only 25% of the time. But observe that this doesn't mean that such sources are not random, just that they aren't *as random* as we would like them to be. Many "true random" number generators contain self-corrector units that will take in the bits from physical processes and do some computation with them to remove the bias. Some of your problems for this week are related to these kind of processes.

3 Making more randomness (a state of sin)

We've seen how a variety of "true" RNGs can be used as a reliable source of entropy. But they also have limitations: they can be "expensive" (in time or in actual cost), and they might be inconsistent or unreliable. At some point, we need a more reliable way to generate random numbers.

What we're talking about here is an *algorithm*, a deterministic routine that spits out as many random numbers as we like, and quickly. Here's what John von Neumann has to say about this endeavor:

> Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin. For, as has been pointed out several times, there is no such thing as a random number — there are only methods to produce random numbers, and a strict arithmetic procedure of course is not such a method.

His point is that *any* method we have to spit out random numbers will not actually be random, because the numbers are perfectly predictable if you know what's inside the memory and how the algorithm works. In other works, these numbers technically have zero entropy. But, as we'll see, it's possible to come up with some very clever ways of generating these so-called *pseudo-random numbers* that will be very useful in practice.

The general set-up of a pseudo-random number generator (PRNG) contains two components:

- The **state** contains memory elements that store all the information needed by the PRNG to produce its values.
- The **procedure** takes the information in the state, runs it through some kind of algorithm, and produces two outputs: the next random number, and the next state.

What this means is that, with an initialized state, the PRNG will happily keep spitting out random numbers forever and ever. Now let's look at some properties and inherent limitations of this process.

The first question to ask is, how do we initialize the state? Nothing can be produced until the state of the PRNG is set to something. Now you might think, why not set it to all zeros or something similar. But because the process is not actually random, each random number output by the PRNG *depends only on the current state.* So if we always initialized the state in the same way, the sequence of numbers would always be identical — very much *not* random!

Instead, we will usually use some outside source of entropy to initialize the state, such as the output of a true RNG as discussed earlier. Other non-random sources (such as the current time) are also used sometimes if security is not an issue. This initial random value used to initialize the PRNG's state is called the *seed.* The analogy is pretty obvious: from this small kernel of randomness we will grow a massive tree of random glory.

Now let's consider the inherent limitations of PRNGs. Say the state is made up of b bits of storage. That means that there are exactly 2^b possible configurations of the state. But then, since the sequence of random numbers generated is only determined by the bits in the state, this means every sequence of PRNG random numbers will be "eventually periodic". That is, eventually, the state will go back into one of its previous settings, and the whole sequence will be identical to what has come before. We know that the maximum period will be 2^b, and one of our tasks in designing PRNGs is to try to come up with efficient procedures which actually come close to achieving the maximum period.

3.1 Linear congruential method

One of the most basic algorithms for PRNGs is called the *linear congruential generator* (LCG). A special case of this method is the *Lehmer LCG*. Here's the procedure.

(Note: I will be using Python syntax to describe algorithms in this class. You shouldn't need to know Python to understand the algorithms, but learning Python might be worth your time. The Python interpreter is called python3 and is free software that is installed on every machine in the Linux labs.)

```
'''NOTE: m and a are parameters for the Lehmer LCG. They are
   "hard-coded" into the algorithm. The seed should be a value
   between 1 and m-1.
'''
def LehmerLCG(seed):
    previous = seed
    while True:
        next = (a * previous) % m
        print(next)
        previous = next
```

Mathematically, we usually write the sequence of random numbers as $X_0, X_1, X_2, \ldots$, where X_0 is the seed value. In this case, the Lehmer LCG is defined by the recurrence relation:

$$X_n = aX_{n-1} \mod m$$

So what about these two parameters m and a? How should they be chosen? Well, we can certainly think of some bad values for a. Try 0 and 1 and see what happens, for example.

There are also some rather bad choices for m. For example, if $m = 30$ and the seed is $X_0 = 10$, then the period of the sequence will be at most 2, no matter what a is. Really, try it! The reason in this case is that 10 is a divisor of 30, which greatly limits the possibilities when we are taking the numbers all mod 30.

Some values of m might not seem quite so bad at first. For example, if we take $m = 25$ and $a = 2$, then choosing a seed value of $X_0 = 16$ gives the following sequence:

$$16, 7, 14, 3, 6, 12, 24, 23, 21, 17, 9, 18, 11, 22, 19, 13, 1, 2, 4, 8, 16, \ldots$$

which has period 20. That's not too shabby. But if we choose a seed of $X_0 = 15$, the sequence is just

$$15, 5, 10, 20, 15, \ldots$$

which has period of only 4. This is very unfortunate behavior, because **the quality of the PRNG should not depend on the seed**. The PRNG designer (that's us!) should choose the parameters m and a so that *any* valid seed will have a long period.

Fortunately, such parameters do exist for the Lehmer LCG. As you might have guessed, m should be a prime number. Then if we make a a primitive root modulo m, then the period will max out at $m-1$, no matter what seed value is chosen. For example, try $m = 17$ with $a = 10$.

One disadvantage of the Lehmer LCG is that the modulus m must be a prime number to get the maximal period, as we have seen. But often, we would really like to generate random numbers from 0 up to $2^k - 1$ for some integer k to generate random bytes or integers. In fact, the simple problem of using a Lehmer LCG to simulate a random coin flip becomes difficult since m must be a prime number.

Another disadvantage of the Lehmer LCG is that it only generates random numbers from 1 up to $m-1$, never giving zeros. To account for this, in practice, we might simply subtract one from each X_i to give pseudo-random numbers in the range of 0 up to $m-2$.

This subtraction idea leads right away to a slightly more sophisticated PRNG that has some nicer properties. It's called the *mixed congruential method*, or sometimes just the linear congruential method (without the "Lehmer" part). Here's pseudocode:

```
'''m, a, and c are parameters.'''
def LCG(seed):
    previous = seed
    while True:
        next = (a * previous + c) % m
        print(next)
        previous = next
```

Notice what changed? Very little! All we did was add c every time to whatever the next number is, and then take mod m of course. Try out some numbers and you will see the advantage here pretty quickly. For instance, with $m = 8$, $a = 5$, $c = 3$, and seed value $X_0 = 2$, we get:

$$2, 5, 4, 7, 6, 1, 0, 3, 2, \ldots$$

The period here is 8, exactly equal to m! This is very exciting, since this means we got every single number in the range from 0 up to $m-1$ exactly once in every period. And what's more, it looks like we can actually choose m to be a power of 2, which is perfect for filling up bytes and words in a computer with random bits.

Of course, there will still be some very poor choices of m, a, and c. But if m is a power of 2, there is a number theory result from Hull and Dobell that proves the following:

> **Theorem**: If $m = 2^k$ for some $k \geq 3$, then the mixed LCG with parameters m, a, and c will have maximal period $m-1$ for any seed value X_0 if and only if:
>
> - $(a-1)$ is divisible by 4, and
> - c is odd

You can confirm that these conditions are satisfied in the example above. Unfortunately, not every a and c gives the same randomness properties, so usually these parameters are chosen very carefully and after much experimentation and statistical testing. For a table with some of the values used in practice, see the Wikipedia page.

The linear congruential generators are nice and quite simple, allowing for extremely fast implementations in software or even in hardware.

However, LCGs are not the most popular PRNG in use today because they do have a few statistical weaknesses. For example, if you use an LCG to generate points in higher dimensions (like setting successive outputs from the PRNG to each dimensional coordinate), the results will not be randomly distributed. This is called a *correlation* because it only appears when we consider the relationships between multiple random points.

In fact, you can see the weakness in LCGs just by thinking about a game of blackjack. If a game of blackjack is dealt from a single deck, a smart player will know, once the deck gets low, which cards are more likely to come next. This is done just by remembering which cards from the 52 possibilities have *not* yet been seen. The same principle can be applied to a LCG because even a very good LCG with a long period will (obviously) not have any repeats in any outputs until the whole sequence repeats itself entirely. This means that, after a long stream of numbers, we know that the next number will *not* be one of those seen recently. For this reason, in practice, the lower order bits of each output from a LCG are often ignored. This eliminates some weaknesses with LCGs such as those just mentioned.

3.2 Linear recurrences

The second weakness we identified in LCGs seems inherent in the process: as long as each output depends only on the previous value, we can't have any repeats in the sequence without the whole thing falling into a cycle. The way to avoid this situation is to base each output on more than just the single previous input.

For example, a very simple kind of generator would just add the previous two values, mod m. This corresponds to the recurrence

$$X_n = X_{n-2} + X_{n-1} \mod m$$

You might recognize this - it's just like the famed Fibonacci sequence, with each one taken modulo p. Unfortunately, this particular sequence turns out to be very predictable and non-random, but a very similar-looking one:

$$X_n = X_{n-24} + X_{n-55} \mod m$$

turns out to be pretty good. And here's another that works well, without having to look back so far into the history:

$$X_n = X_{n-2} + X_{n-3} + X_{n-4} + X_{n-8} \mod 2$$

(Notice that $m = 2$ in this last one.)

What's the common thread between all these? At each step, we update X_n based on the most recent k outputs, possibly multiplying each of them by some constants a_i and then adding all these up modulo m. So overall the equation becomes:

$$X_n = a_1 X_{n-1} + a_2 X_{n-2} + a_3 X_{n-3} + \cdots + a_k X_{n-k} \mod m$$

In all the examples above, the a_i's were all either 0 or 1, but in general they could have any value from 0 up to $m - 1$. But the important thing is that, while the X_i's change every time as n increases, the multipliers a_i never change; they are *constants.*

This kind of a sequence is actually pretty common in multiple areas of computer science and mathematics; it is called a *linearly recurrent sequence.* One of the nice things about such sequences is that they are still pretty simple to implement. The basic algorithm looks like this. (Remember that m and the a_i's are *parameters* of the generator that are hard-coded into the program, decided very carefully beforehand and used over and over again.)

1. Initialize a size-k array with *seed values* in the range of 0 up to $m - 1$.
2. Calculate X_n from the values stored in the array:

X_{n-1}	X_{n-2}	X_{n-3}	$\cdots$	X_{n-k}

3. Print out or return this next value X_n
4. Update the array by "shifting" all elements one position to the right, and adding in X_n, so that now the array contains

X_n	X_{n-1}	X_{n-2}	$\cdots$	X_{n-k+1}

5. Loop back to step 2 *ad infinitum*

Now imagine we want to implement this and make it really fast. What value of m might we choose? Well how about a power of 2? This makes the modulo operation really fast, and it also makes filling up words or bits of memory pretty darn simple.

When $m = 2$, the storage and algorithm can be really easily implemented in hardware, and it's called a linear feedback shift register, or LFSR for short. Go ahead to the Wikipedia page and check it out!

The last thing to discuss is how those magical constants a_i should be chosen. The first requirement we would like is that the maximal period is achieved. Here, the maximal period corresponds to the total number of configurations of the memory, or *state*, of the random number generator. Since it is an array with k values, each from 0 up to $m-1$, the total number of possible memory states is m^k, and this is what we would like the period of the sequence to be.

To determine this mathematically, we define the *characteristic polynomial* of the linearly-recurrent sequence above to be:

$f(x) = x^k - a_1x^{k-1} - a_2x^{k-2} - \cdots - a_{k-1}x - a_k.$

Check out the Wikipedia page on linearly recurrent sequences to see more about this and some of the math behind it. The only thing you really need to know is that *the sequence has maximal period if and only if $f(x)$ is a primitive polynomial mod m.* Your next question is of course what a "primitive polynomial" is. Basically, a primitive polynomial is one where, if we take successive powers of x mod $f(x)$ and mod m, we will eventually get every possible non-zero polynomial with degree less than k. It's very similar to the definition of a "primitive root" mod p that gave the maximal-length Lehmer LCG sequences. Cool!

As always, any choices of these constants will have to be subject to some heavy statistical testing before you would want to use them in practice. But any primitive polynomial will generate a sequence with the maximal period. For example, the sequence defined above with

$$X_n = X_{n-2} + X_{n-3} + X_{n-4} + X_{n-8} \mod 2$$

has characteristic polynomial

$$f(x) = x^8 - x^6 - x^5 - x^4 - 1,$$

which is indeed a primitive polynomial mod 2.

3.3 Mersenne twister and beyond

For many practical purposes, some variation of the linear recurrence generators are "random enough" to use. And because they are so simple, they run very, very quickly and use little memory overhead. The mixed LCGs are even faster and are good enough for many purposes. However, there are some advanced statistical tests that those PRNGs fail unless we set the modulus m and/or the size k to be rather large. Since this affects efficiency, the question becomes whether it's possible to get *excellent* randomness with pretty fast computation times.

The “Mersenne twister” is a PRNG algorithm that was developed by two Japanese researchers in the late 1990s that seems to fit this bill rather well. The basic idea is the same as the linear recurrence generators we already mentioned, except that instead of integers mod m, each array element and each output is an entire *matrix* of numbers modulo 2. The “twist” part is that each output is also multiplied by some fixed matrix (the twist matrix) at the end, to add in another level of scrambling.

The result of this algorithm is a PRNG with an extremely high period that is also very fast to compute. The only downside is that, in order to achieve the high period, the size of the *state* is also relatively large, which means a large amount of seed information is required to start the generator. In most implementations, a separate PRNG (such as a mixed LCG) is used just to generate the seed values, and then the Mersenne twister is run off of that.

The creators of this algorithm actually maintain a website here with many more details and even links to a few popular implementations.

Finally, it must be mentioned that **none of these PRNGs should be used to generate keys for cryptography**. It’s not that they’re not good sources of random-like numbers, just that they might be somewhat predictable if an attacker can, for example, see many of the previously-computed random outputs. Cryptographically-secure PRNGs are an entirely different topic that we’ll touch on later in the class.